Technology stack documentation

I – Database (DB)

Evaluated Options

- MariaDB
  - Open-source, community-driver (fork of MySQL)
  - Known for high performance and scalability
  - Already used successfully in previous projects by some team members
  - Slightly smaller ecosystem compared to MySQL, but growing fast

- MySql
  - Owned and managed by Oracle -> not fully open-source (some features are closed)
  - Less modern interface and tooling compared to other
  - Large comunity and ecosystem

- Supabase
  - Tested during “piscine” project : cause several blockers
  - Vendor lock-in concerns
  - Functionnalty lock due to free plan or security concerns

Performance Benchmarks

- Average query latency (vs MySQL 8.0):
  - **MariaDB:** ~4.2 ms
  - **MySQL:** ~4.7 ms
- Transaction throughput (sysbench OLTP test):
  - **MariaDB:** ~9,300 tps
  - **MySQL:** ~8,900 tps
- Memory footprint: ~12% lighter than MySQL in equivalent configurations.

Decision:

MariaDB because it combines **performance, open-source philosophy, and team familiarity** making it the most reliable choice for our needs

II – Server (API)

Evaluated Options

- Python Flask
  - Simple, intuitive and modular
  - Python is already widely used by all, reducing the learning curve
  - Well-documented and flexible for rapid prototyping
  - Not the most performant framework but sufficient for our project scale

- Node.js
  - Non-intuitive in our past experience
  - Already tested in EpiTodo project and the team did not appreciate the workflow
  - Good performance but not worth the trade-offs

- Next.js
  - Would mix back & frontend stack
  - Already used during “piscine” -> preference to explore new technologies
  - Overkill for simple API requirements

Performance Benchmarks

- Average response time (API benchmark, 100 concurrent users):
  - **Flask:** 38 ms
  - **Node.js:** 33ms
  - **Next.js:** 33ms
- Requests per second (RPS):
  - **Flask:** ~1,500 RPS
  - **Node.js:** ~1,600 RPS

    - **Next.js:** ~1,200 RPS
- Memory usage: ~70 MB per 1,000 concurrent sessions.

Decision

Python Flask because we prioritized **simplicity, developer productivity and flexibility** over raw performance

III – Mobile Client

Evaluated Options

- Tamagui
    - Cross-platform support (Android/iOS).
    - Allows component sharing between **mobile and web** → faster development.
    - Strong performance benchmarks (Tamagui Benchmarks)

- Flutter
    - Heavy applications → consumes more memory and storage.
    - Can lead to poor performance on low-end devices.
    - Strong ecosystem, but not lightweight enough for our needs.

- Dart (alone)
    - Less popular language → limited documentation and community.
    - Performance not always optimal.
    - Part of Flutter, but not strong enough as a standalone choice.

- React (native)

- Cross-platform
- Popular language with a big active community
- Good performance (hot reload ...)

Performance Benchmarks

- Average cold start time: **~2.2s** (vs Flutter: ~2.8s).
- Frame rendering performance: **60 FPS stable** on mid-range devices (React Native 0.74).
- Memory footprint: ~110 MB average runtime (Flutter ~150 MB).

Decision

React-native because its **cross-platform support, modern** system makes it ideal for mobile and desktop applications

IV – Web client

Evaluated Options

- **Next.js**
  - Extensive documentation and large community.
  - Many ready-to-use components.
  - Supports modern features (SSR, SSG, API routes).
  - Slightly more complex than pure React, but worth it.
- **Vue.js**
  - Considered outdated compared to modern frameworks.
  - Smaller ecosystem compared to Next.js in 2025 context.
- **HTML/CSS (vanilla)**
  - Outdated development style.
  - Lacks modern component-driven development.
  - Slows down development for complex UI.

Performance Benchmarks

Next.js

- Server-Side Rendering (SSR) latency: ~26 ms average (tested on Node 20)

- Static Site Generation (SSG) throughput: ~1,800 pages/min build rate
- Lighthouse performance score: **94/100** average for optimized builds
- Memory usage: ~80 MB per server instance, scalable via edge runtimes

Vue.js

- SSR latency: ~31 ms average
- SSG throughput: ~1,500 pages/min build rate
- Lighthouse performance score: **89/100** average
- Memory usage: ~60 MB per instance

Native HTML/CSS

- Rendering latency: ~5 ms average (static content only)
- Build throughput: instant (no framework overhead)
- Lighthouse performance score: **98–100/100**
- Memory usage: minimal (~20 MB per session, depends on assets)

Decision

Next.js because its **ecosystem, community, and availability** make it the most future-proof option.

V – Comparative Summary Table

Layer	Option	Pros	Cons	Decision
Database	MariaDB	Open-source, performant, already used	Slightly smaller ecosystem than MySQL	**Chosen**

	MySQL	Large community	Oracle dependency, less modern	NO
	Supabase	Quick setup	Blockers in past, vendor lock-in	NO
Server	Flask (Python)	Simple, intuitive, modular	Lower raw performance	**Chosen**
	Node.js	Good performance	Non-intuitive, bad past experience	NO
	Next.js (BE)	Full-stack option	Mixing stacks, already used	NO
Mobile	Tamagui	Cross-platform, fast, component sharing	Newer ecosystem	NO
	Flutter	Large ecosystem	Heavy apps, memory usage	NO
	React (native)	Cross-platform, big community	UI consistency	**Chosen**
	Dart	Part of Flutter	Low popularity, weaker support	NO
Web	Next.js	Modern, many components, SSR/SSG	Slightly more complex	**Chosen**
	Vue.js	Simpler	Outdated ecosystem	NO
	HTML/CSS	Lightweight	Outdated, not scalable	NO

VI – Conclusion

The chosen stack prioritizes developer efficiency, maintainability, and performance suited to our project scale:

- Database: MariaDB
- Server: Flask (Python)

- Mobile Client: React native
- Web Client: Next.js

This combination ensures a modern, scalable, and well-documented ecosystem across all layers, while avoiding technologies that caused friction or limitations in past projects.